Artificial Intelligence for Games

Year 2019/20 1st Semester

Group 21

Third project delivery report

Introduction

On This delivery, we were tasked with making decision-making and planning for sir Uthguard. We started out by implementing GOAP and then MCTS. For the MCTS we implemented several variations of the algorithm. We also discuss optimizations that we implemented in our project that improved the overall performance of the game.

GOAP

For this decision-making algorithm, we followed the pseudo-code from the lectures.

Vanilla MCTS

For lab 6 we implemented the MCTS algorithm as presented in the lectures. We used the number of 500-1000 iterations per frame as limit for the MCTS computations. One problem with the vanilla version of MCTS is that if the value for number of iterations per frame is low enough, there will be zero simulations in which sir Uthgard wins. Therefore, all actions are equally “good/bad” for Sir Uthgard and he will select actions arbitrarily. More advanced versions of MCTS handle this problem but for the Vanilla version, the only way to correct this is to increase the number of iterations per frame.

MCTS Biased Playout

For this variation of MCTS we no longer decided actions randomly, but we defined heuristic functions for every action. These functions return a float value that represent how good it is to perform the action given the current game state. We conventionalise that lower H-values corresponds to better actions. Some H-functions depend on the game state, for example picking up a health potion depends on the missing HP of the character and the distance to the potion. Other actions had simpler H-functions, for example Level Up had a constant H-value of -1000 because it is always the best action to do at any given point if it is executable.

Stochastic World

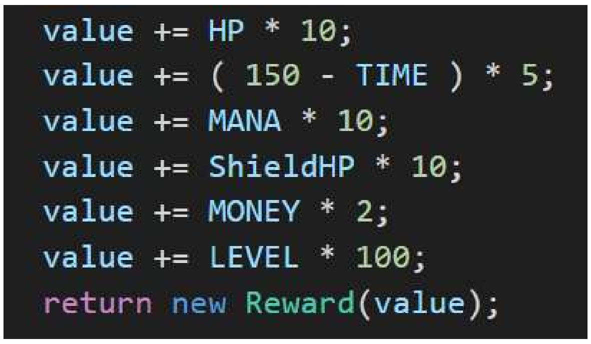
To solve the stochastic world problem, we opted to run multiple playouts (and back propagations) for every selection. This way the agent will simulate the same scenarios more that once and thus will have a better understanding of the expected output of his actions given the randomness of the level.

Optimized World Representation

For this optimization, we opted to use fixed size arrays and attributes. The new world representation worked as a wrapper for the world representation that the code already used. Getters and setters were implemented with switch-case statements that access the arrays.

Limited Playout MCTS

For this variation of MCTS we imposed a hard limit on the number of actions done by playouts. We then defined a function that attributes a score to the current world state. With this new function the agent has much more feedback on how well he is performing. Our new function that aims to point the agent in the right direction is as follows:



```
value += HP * 10;
value += ( 150 - TIME ) * 5;
value += MANA * 10;
value += ShieldHP * 10;
value += MONEY * 2;
value += LEVEL * 100;
return new Reward(value);
```

This algorithm achieved much better results both in terms of win rate and performance. It is worth mentioning the previous scoring function was 0 if the agent lost and 1 if the agent won (apart from some time discounts) which was too simple. This score function will prefer states where the agent is both healthy (HP and Shield HP), has mana (MANA), has progressed through the level (Money and Level) and is being efficient (Time).

Comparisons

Measuring the average total processing time for the different algorithms wielded the following results:

GOAP	MCTS	BP-MCTS	L-BP-MCTS
0,34	0,97	1,15	0,03

Additional Optimizations

In the MCTS algorithm we noticed while debugging that very often children nodes of the same parent will have the same Q-value (both for simple best child selection and UCT). For this reason, we implemented the robust child rule (id two children have the same Q-value, the best child will be the one with a bigger N-value) to decide between children with the same Q value.

Despite having good results in terms of win rate, the Biased versions of MCTS were much slower even when limited in depth. This made the game lag whenever a new decision needed to be made which really hurts the player experience.   We decided to simplify the Heuristic function of the actions as much as possible without hurting the “intelligence” of the agent, namely we modified the *WalkToTargetAndExecuteAction*´s *GetHValue* function to use Manhattan Distance instead of performing a A* search.

Conclusion

In this project we were able to give sir Uthguard, the main character of our game, several different decision-making algorithms that allowed him to complete the level. Giving the agent no information at all about the game (Vanilla MCTS) allowed him to “learn” the best action sequences by himself within manageable computational budget, but the best results in terms of performance were achieved with more sophisticated algorithms that use heuristic functions to guide the agent in a better direction.